Динамический поиск гонок в Java-программах на основе синхронизационных контрактов

Дмитрий Цителов (tsitelov@acm.org), Devexperts LLC

Виталий Трифанов (vitaly.trifanov@gmail.com), СПбГУ

Аннотация. Состояние гонки (data race) возникает в многопоточной программе, когда несколько потоков одновременно обращаются к одному и тому же разделяемому участку памяти, где хотя бы одно обращение – запись. Состояния гонки слабо локализованы и их сложно обнаружить вручную, поэтому исследования в области автоматического поиска гонок ведутся уже более 20 лет. В данной статье рассматриваются вопросы производительности и точности динамического поиска гонок в Java-программах и предлагается идея снижения накладных расходов обнаружения с помощью синхронизационных контрактов. Последние опираются на описания пар вызовов методов, обеспечивающих синхронизацию потоков, и помогают исключать из анализа части целевого приложения, не интересные с точки зрения поиска гонок – например, сторонние библиотеки. Приводится схема языка спецификации контрактов и некоторые технические детали реализации, рассматриваются преимущества и ограничения подхода.

Ключевые слова: многопоточность, состояние гонки, динамический анализ, автоматическое обнаружение ошибок.

1 Введение

С развитием многоядерных и многопроцессорных систем все большее количество программ создаются многопоточными. Использование нескольких потоков выполнения может приводить к дополнительным ошибкам, связанным с некорректной организацией взаимодействия этих потоков. Такие ошибки сложно искать вручную, поскольку чередование операций в потоках обладает высокой степенью неопределённости, и программисту нужно полностью просчитать все возможные ситуации. Одной из типичных ошибок многопоточных программ является наличие в них состояний гонки (data races). Они возникают, когда несколько потоков без должной синхронизации обращаются к одному и тому же разделяемому участку памяти, где хотя бы одно обращение – запись [14]. Дополнительная трудность с обнаружением гонок состоит в том, что они, как правило, не приводят к немедленному сбою и отказу программы. Напротив, приложение продолжает работать с

повреждёнными глобальными структурами данных, что приводит к труднообъяснимым эффектам впоследствии.

За несколько последних десятков лет было разработано несколько статических и динамических подходов к обнаружению гонок. *Статические* детекторы ([7,11,13]) анализируют все пути исполнения программы, не требуют её запуска, но обладают низкой точностью. *Динамические* ([3,4,6,8,15,16,17,18]) анализируют программу во время её работы, но ограничены лишь фактическим путём выполнения программы. Хотя динамические алгоритмы и обладают полной информацией о конкретном пути выполнения программы, на практике их точность ограничена накладными расходами, которые они вносят. Проблема достижения высокой точности обнаружения гонок и отсутствия ложных срабатываний (*false positives*) при сохранении приемлемого уровня потребления аппаратных ресурсов является краеугольной технической проблемой динамического анализа.

Данная работа предлагает способ понижения накладных расходов на динамическое обнаружение гонок в Java-программах без потери точности. Основная идея заключается в сокращении области программы, где отслеживаются все операции синхронизации. Чтобы это не привело к потере информации о синхронизации между потоками и, как следствие, к ложным срабатываниям, вводится понятие синхронизационного контракта, которое позволяет в достаточной степени описать поведение исключённого кода в многопоточной среде. Далее во время работы детектор распознает эти контракты и встретив методы, соответствующие им, обрабатывает эти методы как высокоуровневые синхронизационные события. Такой подход позволяет восполнить неотслеженные операции синхронизации и обеспечить высокую точность поиска.

Дальнейшая часть статьи организована следующим образом. Во второй главе описывается отношение happens-before и приводится формальное определение понятия гонки в терминах спецификации Java, после чего описывается точный алгоритм поиска гонок, основанный на отслеживании этого отношения. Третья глава акцентируется на проблеме обеспечения одновременно точности динамического поиска гонок и приемлемого уровня накладных расходов. В четвертой главе излагается концепция синхронизационных контрактов, в пятой главе – её реализация в разработанном нами динамическом детекторе гонок jDRD. Шестая глава посвящена преимуществам и недостаткам подхода и содержит некоторые экспериментальные результаты. В заключении мы подводим итог и указываем дальнейшие возможные направления работы. В приложении описан язык описания контрактов, используемый в jDRD.

2 Happens-before алгоритм поиска гонок

Отношение happens-before было предложено Лесли Лампортом в работе [10] для установления частичного порядка на множестве всех событий распределенной системы, в которой единственный способ взаимодействия

между участниками – это обмен сообщениями. Данный подход хорошо переносится на многопоточные системы, если рассматривать каждый поток как отдельного участника системы, а операции синхронизации – как передачу сообщений. Язык программирования Java был одним из первых языков с собственной архитектурно-независимой моделью памяти, определяющей семантику взаимодействия потоков через отношение happens-before, и распространяющей это отношение на все операции в программе.

Для этого в спецификации Java [9] сначала вводится отношение *синхронизированности* (synchronized-with) – полное отношение порядка на множестве всех операций синхронизации в программе. Так,

- освобождение монитора *синхронизировано* с его последующими захватами;
- запись volatile-переменной *синхронизирована* с ее последующими чтениями;
- запуск потока *синхронизирован* с первым действием в потоке;
- последнее действие в потоке T_1 *синхронизировано* с любым действием в другом потоке, случившемся после того, как тот обнаружил, что T_1 завершился;
- прерывание потока T_1 *синхронизировано* с любым действием в другом потоке, которое обнаружит, что T_1 получил сигнал прерывания.

События слева (например, захват монитора) будем называть *отправкой*, а справа (отпускание монитора) – *приёмом* синхронизационного сообщения. Обратим внимание, что во всех случаях с синхронизационным сообщением естественным образом ассоциирован уникальный объект – монитор, volatile-переменная или объект потока. Поэтому далее будем считать, что с каждой парой синхронизированных событий связан некий уникальный *синхронизационный объект*.

Объединение отношения синхронизированности с естественным темпоральным отношением порядка в рамках операций одного потока даёт частичное отношение порядка – отношение *предшествования* (happens-before), определённое уже на множестве всех операций в программе. Факт «событие x произошло перед событием y» записывается как *hb(x,y)*. Само отношение определяется следующим образом:

- если x и y – операции в одном потоке, и x произошло раньше y, то hb(x,y);
- завершение конструктора объекта предшествует запуску его финализатора;
- если события x и y синхронизированы, то hb(x,y);
- транзитивность: если hb(x,y) и hb(y,z), то hb(x,z).

Отслеживание отношения happens-before позволяет определить, получил ли некоторый поток информацию о действиях другого потока – например, изменения в разделяемой памяти, новые значения переменных и т.д.

Наконец, спецификация Java формально определяет понятие гонки: два обращения x и y к разделяемой переменной из различных потоков, хотя бы одно из которых – запись, образуют гонку, если ни одно из них не предшествует другому:

$$DR(x,y) \Leftrightarrow !hb(x,y) \wedge !hb(y,x).$$

Таким образом, для точного обнаружения гонок в программе достаточно отслеживать конфликтующие обращения различных потоков к разделяемым переменным, не упорядоченные отношением happens-before. Для этого традиционно используются *векторные часы* (vector clock)[1], предложенные в работе [12]. Векторные часы представляют собой динамический массив неотрицательных чисел, равный по размеру количеству потоков в программе. Над векторными часами определена операция слияния:

$$merge(V_1, V_2): \text{for each } i \in [1..n]\ V_1[i] := \max(V_1[i], V_2[i])$$

Каждый поток хранит свои векторные часы, отражающие его знания о системе: i-я компонента его часов равна последней компоненте i-го потока, которые он наблюдал[2]. Компонента, соответствующая ему самому, называется *собственной* компонентой. Изначально собственная компонента часов потока проинициализирована единицей, а остальные – нулями. Перед каждой операцией синхронизации x в потоке T_1, собственная компонента часов потока T_1 увеличивается на единицу, а часы загружаются в часы соответствующего синхронизационного объекта. Когда в другом потоке T_2 происходит событие y, такое что x синхронизированно с y, в часы потока T_2 загружаются часы синхронизационного объекта. Это позволяет отследить отношение synchronized-with.

Для отслеживания отношения happens-before нужно проассоциировать векторные часы с каждой разделяемой переменной в программе. Когда поток T_1 обращается к такой переменной, он покомпонентно сравнивает свои часы с часами переменной. Если есть компонента часов потока, которая не превосходит соответствующей компоненты часов переменной, обнаружена гонка. Далее поток загружает свои часы в часы переменной.

Если в программе N потоков, то хранение одних векторных часов требует O(N) памяти, и основные операции над ними также имеют сложность O(N). В работе [8] показано, что для отслеживания отношения happens-before вместо полных часов переменной достаточно хранить номер и собственную компоненту потока, который последним обращался к этой переменной. Таким образом, значительная часть операций будет требовать O(1) и снизится количество потребляемой памяти, что позволило алгоритму happens-before достичь уровня производительности менее точных динамических алгоритмов.

3 Точность поиска и накладные расходы

К сожалению, точечные оптимизации алгоритма, хотя и производят безусловный положительный эффект (что подтверждает множество исследований), не могут обеспечить приемлемый уровень накладных расходов

---

[1] Эквивалентное название – *логические часы* (logical clocks).

[2] Не умаляя общности, предполагается, что все потоки пронумерованы от 1 до n.

на приложениях, состоящих хотя бы из нескольких тысяч классов и использующих 10-20 потоков.[3] Подавляющее большинство исследовательских работ останавливается на прототипе детектора и модельном тестировании. Подробный обзор состояния предметной области можно найти в работах [1,2]. Нам удалось обнаружить лишь два готовых к использованию детектора (IBM MSDK[16] и TSan[18]), но попытки использовать их на крупных приложениях приводили либо к немедленному отказу, либо к переполнению памяти. О схожих проблемах сообщают также авторы работы [15].

Для обеспечения точности поиска нужно отслеживать все операции обращения к разделяемым данным и все операции синхронизации. Первые можно ограничить путём отсечения областей кода, в которых нас не интересуют гонки – например, сторонние библиотеки или стандартные Java-классы. Для большинства разрабатываемых программных систем решения об использовании сторонних компонент принимаются на стадии проектирования, когда уже известны требования по надежности и безотказности системы, поэтому есть смысл считать, что сторонние компоненты обладают достаточной степенью надежности и концентрироваться на обнаружении ошибок в непосредственно разрабатываемом программном коде и проверке корректности использования сторонних компонент.

Однако исключить таким же образом операции синхронизации нельзя, поскольку их пропуск приведет к неполучению информации о передаче отношения happens-before и, как следствие, к ложным срабатываниям, – детектор будет сигнализировать о гонках, которых в действительности не произошло. Количество этих операций велико и экспоненциально растёт с увеличением количества потоков в программе, что в совокупности с необходимостью отслеживать операции обращения к разделяемым данным в итоге и приводит к описанным эффектам.

4 Синхронизационные контракты

Принцип инкапсуляции в объектно-ориентированном подходе к программированию, которому следует Java, предполагает, что использование объекта осуществляется посредством вызова его публичных методов. Поэтому, в идеале, достаточно иметь возможность описать правила использования методов исключённых классов в многопоточной среде. Возможны следующие варианты:

- метод не потокобезопасен, то есть его одновременное использование несколькими потоками не предусмотрено и требует внешней синхронизации;
- метод потокобезопасен; он может быть вызван (и в некотором смысле предназначен для этого) несколькими потоками одновременно без внешней синхронизации.

[3] Обратим внимание, что характеристики многих промышленных программных систем могут в десятки и тысячи раз превосходить указанные.

Вызовы методов первого типа необходимо отслеживать и обрабатывать по алгоритму happens-before как обращения к объекту-владельцу метода на чтение, если метод немодифицирующий, или на запись в противном случае. Разработанный нами детектор jDRD трактует по умолчанию все обращения как модифицирующие и предоставляет возможность на уровне конфигурации указать, что некоторые конкретные методы являются немодифицирующими.

С вызовами методов второго типа немного сложней. Вообще говоря, их можно игнорировать, поскольку они предназначены для многопоточного использования. Однако именно в эту группу методов попадают те, которые обеспечивают синхронизацию между потоками. Как правило, такие методы содержатся в классах, созданных как средство обеспечения корректного взаимодействия потоков, что чётко декларировано в их описании. Так, начиная с версии 1.5 в Java появились высокоуровневые средства синхронизации, отличные от захватов/блокировок мониторов и volatile-переменных. В основном они содержатся в пакете java.util.concurrent и его подпакетах.

Например, там есть потокобезопасная хеш-таблица ConcurrentHashMap, которая гарантирует, что вызов метода put по некоторому ключу *предшествует* последующему вызову метода get на том же объекте по тому же ключу [5].

Более формально, это означает, что между событиями вызова метода put в первом потоке и метода get во втором потоке есть цепочка событий, через которые передается отношение happens-before. Если детектор отслеживает все операции синхронизации, то он обнаружит эту цепочку и вычислит, что эти два вызова методов также упорядочены отношением happens-before.

Теперь предположим, что мы хотим исключить класс ConcurrentHashMap из области анализа – то есть, не отслеживать в нем операции синхронизации. В этом случае детектор не получит информации о том, что вызов put предшествует вызову get. Таким образом, ему её нужно явно сообщить. Отметим, что для этого не подходят аннотации, поскольку в случае библиотечных классов нет возможности модифицировать исходный код. Нами был разработан метод конфигурирования на базе языка xml.

Рассмотрим два метода: метод $f(P_{11}, \dots P_{1n})$ объекта O_1 и метод $g(P_{21}, \dots, P_{2m})$ объекта O_2, где $n,m \geq 0$. Объект O_1 будем называть объектом-владельцем метода f, а O_2 – объектом-владельцем метода g. Между этими методами может быть *примитивная явная связь* одного из трёх типов:

1. связь «владелец-владелец»: $O_1 == O_2$, то есть методы принадлежат одному объекту;
2. связь «владелец-параметр»: $n > 0$, $\exists i \in [1..n]$: $O_2 == P_{1i}$ или $m > 0, \exists j \in [1..m]$: $O_1 == P_{2j}$, то есть параметр одного метода является объектом-владельцем другого метода;
3. связь «параметр-параметр»: $n,m > 0$, $\exists i \in [1..n]$, $j \in [1..m]$: $P_{1i} == P_{2j}$, то есть i-й параметр метода f и j-й параметр метода g являются одним и тем же объектом.

Будем называть *явной связью* комбинацию любого количества примитивных связей.

Будем называть *синхронизационным контрактом* описание пары *явно связанных* методов, которые, будучи вызванными в определённом порядке, гарантируют синхронизацию потоков. Рассмотрим пример с классом ConcurrentHashMap. Синхронизационный контракт на методы put и get – это явная связь, образованная из связей первого (речь идёт о методах одной и той же map) и третьего (должен быть один и тот же ключ – первый параметр каждого метода) типа. Описание этого контракта на языке конфигурирования jDRD представлено ниже. Подробное описание языка см. в приложении.

```
<Sync>
    <Links>
        <Link send="owner" receive="owner"/>
        <Link send="param" send-number="0" receive="param" receive-
number="0"/>
    </Links>
    <Send>
        <MethodCall owner="java.util.concurrent.ConcurrentMap"
name="put" descriptor="(Ljava/lang/Object;Ljava/lang/Object;)
Ljava/lang/Object;"/>
    </Send>
    <Receive>
        <MethodCall owner="java.util.concurrent.ConcurrentMap"
name="get" descriptor="(Ljava/lang/Object;)Ljava/lang/Object;"/>
    </Receive>
</Sync>
```

Рис. 1. Пример синхронизационного контракта.

В качестве примера примитивной связи второго типа можно привести контракт метода execute класса Executor, обеспечивающий асинхронное выполнение задачи. Он принимает в качестве параметра объект типа Runnable и впоследствии вызывает его метод run в другом потоке. Спецификация метода execute гарантирует, что его вызов *предшествует* последующему вызову метода run объекта, переданного в метод execute в качестве параметра.

В следующем разделе мы представим наш детектор jDRD и реализацию синхронизационных контрактов в нём.

5 Реализация

Для отслеживания различных операций в программе необходимо внедриться в ход её выполнения и перехватывать обращения к методам, переменным и т.д. В Java это традиционно реализуется на уровне байт-кода. Это удобно, поскольку он хорошо структурирован и вместе с тем достаточно просто организован. Кроме того, в отличие от исходного кода, байт-код классов доступен всегда.

JVM (виртуальная машина Java) позволяет подключить к ней компоненту, реализующую особый интерфейс *java-агента*, который будет получать управление перед загрузкой очередного нового класса. Агенту передаётся массив байт, содержащий байт-код загружаемого класса, который агент может *трансформировать*. В jDRD реализован этот интерфейс в отдельной компоненте. jDRD-агент анализирует байт-код загружаемых классов с помощью библиотеки ASM, находит интересующие инструкции (захват/освобождение монитора, обращение к разделяемой переменной, запуск потока и т.д.) и вставляет после них соответствующие внутренние вызовы детектора.

Таким образом jDRD получает информацию об операциях синхронизации и обращениях к разделяемым данным в программе. Далее он обрабатывает их по алгоритму happens-before.

Остановимся подробней на обработке операций синхронизации. Когда jDRD встречает событие отправки синхронизационного сообщения, он должен увеличить собственную компоненту часов потока на единицу. После этого ему нужно сохранить свои часы так, чтобы они были доступны другому потоку, в котором произойдет соответствующее событие приёма синхронизационного сообщения. Как отмечалось выше, с каждым синхронизационным событием можно связать уникальный синхронизационный объект. jDRD пользуется этим фактом, и сохраняет часы потоков в хеш-таблице, ключами в которой являются те самые синхронизационные объекты. Для события освобождения монитора таким ключом будет ссылка на объект-владелец монитора, а для volatile-переменной – пара (название переменной, объект-владелец переменной). [4]

Перейдем к отслеживанию синхронизационных контрактов. Их нужно трактовать как высокоуровневые операции синхронизации и ассоциировать с этими операциями искусственный синхронизационный объект. Поскольку описание контрактов содержится в отдельном xml-файле, они читаются и анализируются после запуска детектора. Далее для каждого контракта динамически генерируется класс, сущности которого впоследствии будут использоваться как синхронизационные объекты для данного контракта. Например, для контракта с рис. 1 будет создан следующий класс:

```
class CompositeKey1 {
    Object o1; //для примитивной связи владелец-владелец
    Object o2; //для примитивной связи параметр-параметр
}
```

Когда jDRD встретит вызов метода ConcurrentMap.put, он обнаружит, что этот метод является частью синхронизационного контракта. В этот контракт входят две примитивные связи, каждая из которых опирается на Object. jDRD отыщет соответствующий класс-ключ (в данном случае это класс CompositeKey1) и

[4] Для предотвращения утечек памяти в хеш-таблице используются не обычные (*сильные, strong reference*) ссылки на ключи, а слабые (*weak reference*). Ключевое свойство последних заключается в том, что если на объект остаются только слабые ссылки, то он может быть удалён сборщиком мусора.

создаст новый объект этого типа, который и будет синхронизационным объектом для данного исполнения контракта. Далее часы потока будут увеличены на единицу и записаны в хеш-таблицу. После этого jDRD перестаёт отслеживать операции синхронизации до того момента, когда метод put завершится. Чтобы распространить информацию о том, что отслеживание операций синхронизации в данном потоке стоит приостановить, в векторных часах потока есть переменная-флаг, которая выставляется в true, когда поток входит внутрь контрактного метода и сбрасывается обратно в false при выходе из него. Если контрактный метод вызывается с уже установленным флагом, то флаг оставляется без изменений. Таким образом, сохраняется возможность корректной работы с контрактами различного уровня, часть из которых может быть использована при реализации других. При перехвате любой операции синхронизации jDRD в первую очередь проверяет состояние флага, и если оно равно true, он просто игнорирует эту операцию.

6 Преимущества и ограничения подхода

Алгоритм обнаружения гонок, применяемый в jDRD, базируется на учёте всех отношений happens-before для выяснения корректности производимой операции доступа к данным. Поскольку отношения happens-before при выполнении операций синхронизации, заданные моделью памяти Java, «тотальны» (устанавливают отношение со всеми операциями, предшествующими точке синхронизации), любое событие синхронизации в коде программы имеет неограниченное воздействие на все отслеживаемые переменные и взаимодействующие потоки.

Задача детектора в идеальном случае – проверить корректность исполняемого кода только с учётом явно обозначенных контрактов используемых библиотек. Построение отношения happens-before с учётом всех произошедших операций синхронизации для кода, контракт которого не задаёт явных условий синхронизации, может привести к тому, что код, который с формальной точки зрения недостаточно синхронизирован, будет считаться корректным при рассмотрении всех промежуточных операций. Например, вызов стандартного java-метода System.out.err.print (используется вывода сообщения об ошибке) содержит внутри себя критическую секцию. Если два потока вызывают этото метод поочередно, они синхронизируются между собой, но это является деталью реализации этого метода, побочным эффектом, а не декларированным свойством. Подобные синхронизационные события не только не представляют интереса, но и создают дополнительный «шум», затрудняющий обнаружение гонок.

Таким образом, производимые на основе описанных контрактов изъятия промежуточных синхронизационных событий позволяют не только уменьшить общий объём работы анализатора, но и увеличить вероятность обнаружения гонок в некорректно синхронизированном коде, т.е. повышают точность метода.

Это подтверждается нашими лабораторными экспериментами, которые проводились на трёх приложениях:

- JTT – пользовательское клиентское приложение к системе отслеживания ошибок – порядка 400 классов, 10 потоков;
- QDTest – нагрузочный тест системы распространения котировок – 700 классов, 15 потоков;
- MARS – крупная многоцелевая мониторинговая система – 2000 классов, 30 потоков.

Мы запускали детектор jDRD на нескольких приложениях в двух режимах:

- *базовый режим* – отслеживание операций синхронизации во всём коде программы без использования каких-либо синхронизационных контрактов;
- *juc-режим* – описаны контракты для всех классов пакета java.util.concurrent и для класса sun.misc.Unsafe.

Краткие результаты этих запусков представлены в таблице 1, из которой видно, что использование контрактов снизило количество хранимых векторных часов и количество обрабатываемых операций синхронизации. Это, в свою очередь, снизило нагрузку на приложение и позволило обнаружить больше гонок. Все гонки, обнаруженные в базовом режиме, были также обнаружены и в juc-режиме, что свидетельствует о повышении точности поиска гонок.

	JTT		**QDTest**		**MARS**	
Режим	базовый	juc	базовый	juc	базовый	juc
Синхр. оп-ий/сек	115K	28K	7400K	4300K	1650K	800K
Кол-во часов	13K	7K	85K	72K	15K	14K
Найдено гонок	8	10	1	5	3	7

Таблица 1. Количество обрабатываемых синхронизационных операций в секунду и количество хранимых векторных часов в различных режимах работы jDRD.

С другой стороны, подход обладает рядом ограничений.

Во-первых, с помощью предложенных синхронизационных контрактов возможно описание лишь явно связанных методов. Можно представить себе и неявную связь, но это скорее исключение, чем правило. В пакете java.util.concurrent есть несколько таких методов, например метод newCondition объекта типа Lock. Этот метод возвращает объект типа Condition, внутренне связанный с исходным объектом типа Lock. В этом случае детектор просто «шагнет» внутрь этих классов и будет продолжать анализ.

Во-вторых, jDRD трактует методы, являющиеся частью синхронизационных контрактов, как атомарные, в то время как они таковыми не являются. Операция в программе, которая непосредственно обеспечивает синхронизацию потоков, обычно находится где-то внутри метода и может быть существенно отделена по времени от точек входа и выхода из него. Следовательно, на момент завершения

работы метода информация может быть устаревшей, что может привести к ложным срабатываниям.

7 Заключение

Одной из принципиальных проблем динамического анализа программ является обеспечение сочетания точности и глубины анализа и приемлемого уровня накладных расходов. В задаче автоматического поиска гонок эта проблема особенно актуальна, поскольку количество операций синхронизации и обращений к разделяемым данным очень велико. Если область программы, в которой необходимо отслеживать обращения к разделяемым переменным является скорее вопросом конфигурации и выделения наиболее «интересных» и опасных участков кода, то операции синхронизации нужно отслеживать во всем коде программы, чтобы не пропустить информацию о синхронизации потоков.

В данной работе мы предлагаем концепцию синхронизационных контрактов, в основе которой лежит идея отслеживания не всех операций синхронизации, происходящих в программе, а лишь явно декларированных. Мы вводим понятие синхронизационного контракта как пары явно связанных методов и предлагаем язык для их описания, основанный на xml-нотации. Поддержка и корректная обработка описанных таким образом контрактов была реализована нами в детекторе jDRD, который посредством трансформирования байт-кода внедряется в java-программу, отслеживает в ней важные события (в том числе и методы, описанные в контрактах) и обрабатывает их по алгоритму happens-before. Лабораторное тестирование показывает эффективность использования контрактов – сокращается как количество обрабатываемых операций синхронизации, так и количество векторных часов, которые необходимо хранить для отслеживания отношения happens-before.

Подход обладает рядом ограничений, над устранением которых мы планируем продолжить работу. Также мы проводим внедрение jDRD в процесс разработки ПО и промышленную апробацию, результаты которой надеемся опубликовать в дальнейшем.

Список литературы

1. Трифанов В.Ю., Цителов Д.И. Динамические средства обнаружения гонок в параллельных программах. Компьютерные инструменты в образовании, №5, 2011. С. 3-15.
2. Трифанов В.Ю., Цителов Д.И. Статические и post-mortem средства обнаружения гонок в параллельных программах. Компьютерные инструменты в образовании, №6, 2011. С. 3-13.
3. Choi J., Lee K., Loginov A., O'Callahan R., Sarkar V., Sridharan M. Efficient and precise datarace detection for multithreaded object-oriented programs. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, 2002. P. 258–269.

4. Christiaens M., Brosschere K. TRaDe: A topological approach to on-the-fly race detection in Java programs. In Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium, Vol. 1, 2001. P. 105–116.
5. Documentation of java.util.concurrent package, http://download.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html
6. Elmas T., Qadeer S., Tasiran S. Goldilocks: A Race and Transaction-Aware Java Runtime. In Proceedings of The 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07), 2007. P. 245–255.
7. Engler D., Ashcraft K. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In Proceedings of The Nineteenth ACM Symposium on Operating Systems Principles, 2003. P. 237–252.
8. Flanagan C., Freund S. FastTrack: Efficient and Precise Dynamic Race Detection. In ACM Conference on Programming Language Design and Implementation, 2009. P. 121–133.
9. Java Language Specification, Third Edition. Threads and Locks. Happens-before Order. http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.5
10. Lamport L. Time, Clocks and the Ordering of Events in a Distributed System. Communications of the ACM, Vol. 21, Issue 7, 1978. P. 558–565.
11. Leino K., Nelson G., Saxe J. ESC/Java user's manual. SRC Technical Note 2000–002, 2001.
12. Mattern, F. Virtual Time and Global States of Distributed Systems. In Cosnard, M., Proc. Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, France: Elsevier, pp. 215–226, 1988.
13. Naik M., Aiken A., Whaley J. Effective Static Race Detection for Java. In Proceedings of The 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2006. P. 308–319.
14. Netzer R., Miller B. What Are Race Conditions? Some Issues and Formalizations. In ACM Letters On Programming Languages and Systems, 1(1), 1992. P. 74–88.
15. Pozniansky E., Schuster A. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In Proceedings of The Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2003. P. 179–190.
16. Qi Y., Das R., Luo Z., Trotter M. MulticoreSDK: a practical and efficient data race detector for real-world applications. In Proceedings Software Testing, Verification and Validation (ICST), IEEE, 21-25 March 2011. P. 309–318.
17. Savage S., Burrows M., Nelson G., Sobalvarro P., Anderson T. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. In ACM Transactions on Computer Systems, Vol. 15, Issue 4. , 1997. P. 391–411.
18. ThreadSanitizer for Java: a run-time detector of data raceshttp://code.google.com/p/java-thread-sanitizer/

Приложение. Язык описания синхронизационных контрактов.

Для описания happens-before контрактов пар методов различных классов предназначен тег `Syncs`, содержащий несколько тегов `Sync`, - по одному на каждый контракт. В описании контракта указываются все примитивные связи, образующие связь между методами (тег Links, содержит по одному тегу на каждую примитивную связь), и описание вызовов методов, являющихся

отправкой и приёмкой отношения happens-before (теги `Send` и `Receive`, соответственно).

```
<!ELEMENT Syncs ( Sync+ ) >
<!ELEMENT Sync ( Links, Send, Receive ) >
<!ELEMENT Receive ( MethodCall ) >
<!ELEMENT Send ( MethodCall ) >
<!ELEMENT Links ( Link+ ) >
```

Вызов метода описывается в теге `MethodCall`: указывается класс-владелец метода, название метода и его дескриптор во внутренней нотации виртуальной Java-машины (JVM).

Примитивная связь описывается тегом `Link`:

```
<!ELEMENT Link EMPTY >
<!ATTLIST Link
  receive (owner|param) #REQUIRED >
  receive-number CDATA #IMPLIED >
  send (owner|param) #REQUIRED >
  send-number CDATA #IMPLIED >
```

Атрибуты `send` и `send-number` соответствуют левой части примитивной связи, а `receive` и `receive-number` – правой. Обе они могут быть либо типа «владелец», либо типа «параметр». Если правая часть типа «владелец», то `receive` имеет значение «owner», а `receive-number` не указывается. В другом случае `receive` имеет значение «param», а `receive-number` содержит номер параметра в сигнатуре метода (нумерация начинается с нуля). Аналогично для атрибутов `send` и `send-number`. Пример такого контракта приведён выше на рис.1.

Если нужно описать контракты нескольких пар методов одного и того же класса, это можно сделать с помощью тега `Multiple-Syncs`, состоящего из нескольких тегов `Multiple-Sync`, каждый из которых соответствует одному классу. Полное имя класса указывается в атрибуте `owner`, а связь между вызовами методов описывается в теге `Multiple-Links`:

```
<!ATTLIST Multiple-Sync owner ID #REQUIRED >
<!ELEMENT Multiple-Syncs ( Multiple-Sync+ ) >
<!ELEMENT Multiple-Sync ( Multiple-Links, Call+ ) >
<!ELEMENT Multiple-Links ( Multiple-Link+ ) >
```

Пример такого контракта приведён на рис. 2.

```
<Multiple-Sync owner="java.util.concurrent.atomic.AtomicBoolean">
     <Multiple-Links>
         <Multiple-Link type="owner"/>
     </Multiple-Links>
     <Call type="receive" name="get" descriptor="()Z"/>
```

```
    <Call type="full" name="compareAndSet" descriptor="(ZZ)Z"
          shouldReturnTrue="true"/>
    <Call type="send" name="set" descriptor="(Z)V"/>
    <Call type="full" name="getAndSet" descriptor="(Z)Z"/>
</Multiple-Sync>
```

Рис. 2. Пример описания синхронизационных контрактов для нескольких методов одного класса.